GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNOLOGY

Lexical Analyzer MiniLang

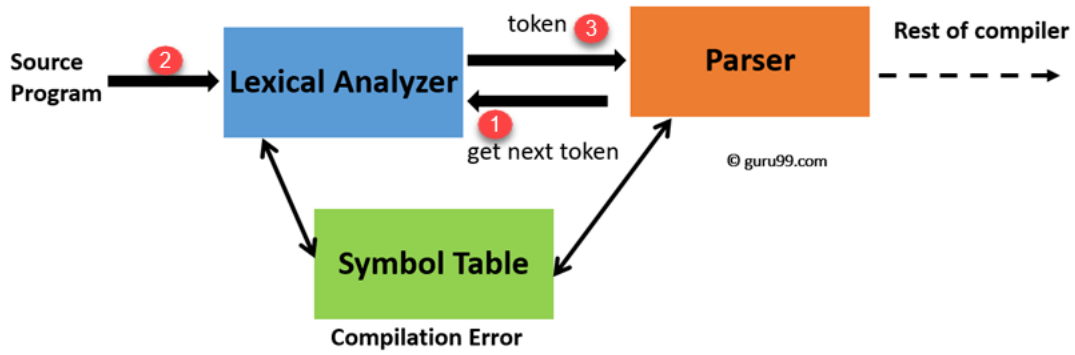
CS424 (CS) Compiler Construction - Assignment 2

Submitted By: 2020519 Wardah Tariq

Submitted to: Sir Usama Janjua

I. Introduction

The goal of this project is to design and implement a parser for a simple programming language named "MiniLang." MiniLang supports basic programming constructs such as arithmetic expressions, variable assignments, if-else conditions, and print statements. This report outlines the design decisions, implementation details, and documentation of the MiniLang parser.



II. Scanner

The scanner takes as input a source file that contains the contents that we want to read. The contents of the file are sent to our scanner also known as the lexical analyzer which then reads the contents of the file and breaks it down into lexemes. Each lexeme contains the token along with the detected tokens value.

Our tokens will be categorized into 7 different types shown below:

1. INTEGER
2. BOOLEAN
3. IDENTIFIER
4. OPERATOR = *, +, /, -, ; , =
5. KEYWORD = Print, If, Else, True, False
6. COMMENT = #
7. END_OF_FILE

III. Parser

A parser is responsible for transforming input tokens, generated through lexical analysis, into an Abstract Syntax Tree (AST). The AST is constructed by analyzing the syntactic structure of the code, adhering to the rules specified by the programming language's grammar. This tree-like structure portrays the hierarchical arrangement of the code, capturing syntactic constructs and their relationships. It serves as a fundamental representation of the code's syntax, facilitating subsequent analysis and transformations during compilation or interpretation.

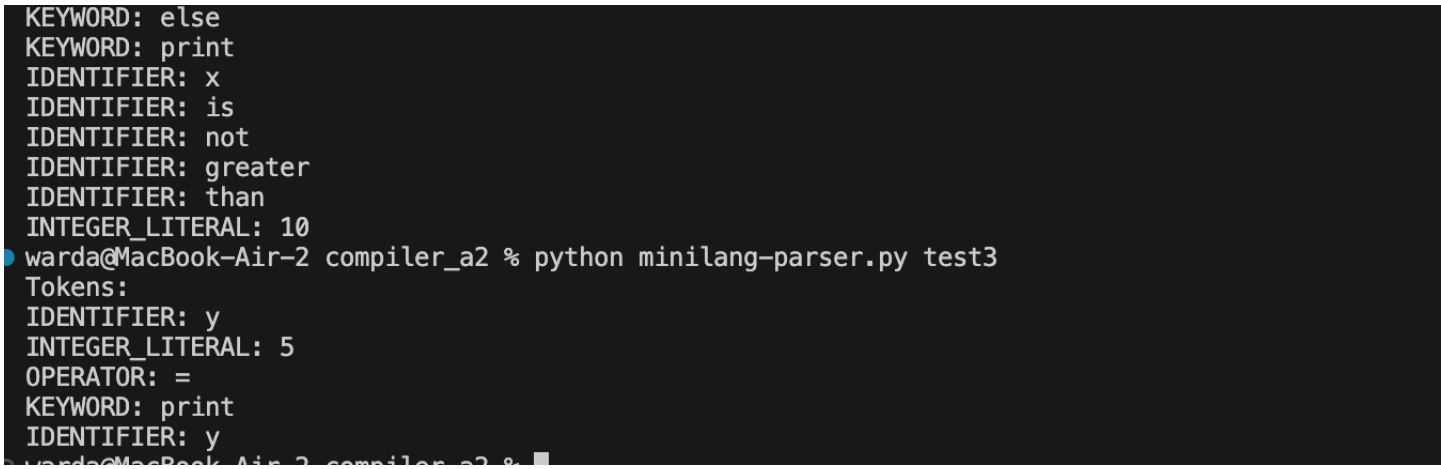
Our implemented parser systematically traverses the tokens, either iteratively or recursively, to validate whether the provided stream of tokens adheres to the language's syntax rules. If an unexpected token is encountered, signaling a deviation from the expected structure, the parser identifies a syntax error and communicates this finding. This process ensures that the parser effectively identifies and reports syntax errors while analyzing the stream of tokens.

Below we can see all the production rules of our context free grammar that we have defined in our parser class:

- **program** → <statement> | END_OF_FILE
- **statement** → <assignment> | <conditional> | <printStatement>
- **assignment** → = <expression> ;
- **conditional** → if (<expression>) { <program> } | if (<expression>) { <program> } else { <program> }
- **printStatement**> → PRINT <expression> ;
- **expression** → <term> + <term> | <term> - <term> | <term>
- **term** → <factor> * <factor> | <factor> / <factor> | <factor>
- **factor** → INTEGER | IDENTIFIER | (<expression>)

III. Test cases

```
warda@MacBook-Air-2 compiler_a2 % python minilang-parser.py test1
Tokens:
IDENTIFIER: x
OPERATOR: =
INTEGER_LITERAL: 20
KEYWORD: print
IDENTIFIER: x
warda@MacBook-Air-2 compiler_a2 % python minilang-parser.py test2
Tokens:
IDENTIFIER: x
OPERATOR: =
INTEGER_LITERAL: 2
KEYWORD: if
IDENTIFIER: x
INTEGER_LITERAL: 10
KEYWORD: print
IDENTIFIER: x
IDENTIFIER: is
IDENTIFIER: greater
IDENTIFIER: than
INTEGER_LITERAL: 10
```


KEYWORD: else
KEYWORD: print
IDENTIFIER: x
IDENTIFIER: is
IDENTIFIER: not
IDENTIFIER: greater
IDENTIFIER: than
INTEGER_LITERAL: 10
warda@MacBook-Air-2 compiler_a2 % python minilang-parser.py test3
Tokens:
IDENTIFIER: y
INTEGER_LITERAL: 5
OPERATOR: =
KEYWORD: print
IDENTIFIER: y